

METHOD AND SYSTEM FOR OPTIMIZING LAYERED COMMUNICATION
PROTOCOLS

CROSS-REFERENCE TO A RELATED APPLICATION

5

This application claims priority on earlier filed provisional patent application Serial No. 60/057,602, filed August 30, 1997, which is incorporated herein by reference.

10

1. Field of the invention.

15

The present invention relates to transmitting data over digital networks, and, in particular, to decreasing actual computation and layering overhead in addition to improving latency and increasing throughput by reducing overhead caused by interfaces and headers in different protocol layers.

20

2. Background art

25

Distributed systems employ communication protocols for reliable file transfer, window clients and servers, RPC atomic transactions, multi-media communication, etc. Layering of protocols has been known as a way of dealing with the complexity of computer communication. Layered protocols offer such significant advantages as developing and testing high-level protocols broken into small layers more rapidly than large monolithic non-layered protocols. Layered protocols are modular and can often be combined in various ways, allowing the application designer to add or remove layers depending on the properties required. In many layered systems where different protocols are substitutable for one

30

Ins. >
B.

09667439-101300

another, application designers can select a combination of protocols most suited to their expected work load. In addition, systems such as Ensemble support changing protocol stacks underneath executing applications, so
5 the application can tune its protocol stack to its changing work load.

Unfortunately, the convenience of having a stack of protocols is often overshadowed by the problem that
10 layering produces a lot of overhead which, in turn, increases delays in communication. Extensively layered group communication systems where high-level protocols are often implemented by 10 or more protocol layers greatly reduce design complexity of a communication
15 network. On the other hand, extensive layering often leads to serious performance inefficiencies.

The disadvantages of layered systems leading to performance inefficiencies consist primarily of overhead, both in computation and in message headers,
20 caused by the abstraction barriers between layers. Because a message often have to pass through as many as 10 or more protocol layers on its way from a host to the network and from the network to a host, the overhead produced by the boundaries between the layers is often
25 more than the actual computation being done. Different system have reported overheads for crossing layers of up to 50 μ s. Therefore, it is highly desirable to mitigate the disadvantages and to develop techniques that reduce delays by improving performance of layered protocols.

30 Several methods have been suggested to improve performance of layered communication protocols. One of the methods is described by Robbert van Renesse in the article "Masking the Overhead of Protocol Layering", Proc. of the Proceedings of the 1996 ACM SIGCOMM

09687439 101300

The described protocol accelerator optimization model successfully reduces communication latency, but
30 does not decrease actual computation and layering overhead. It would also be desirable to optimize a larger class of communication protocols, including outing and total ordering protocols. Moreover, the protocol accelerator approach requires structural

The described protocol accelerator optimization model successfully reduces communication latency, but
30 does not decrease actual computation and layering overhead. It would also be desirable to optimize a larger class of communication protocols, including outing and total ordering protocols. Moreover, the protocol accelerator approach requires structural

modifications to protocols that are effectively annotations. It would be desirable to employ such optimization that calls for significantly less annotation.

5 Other work on protocol optimization has been done
on Integrated Layer Processing (ILP) in "Analysis of
Techniques to Improve Protocol Processing latency; in
Proc. of the Proceedings of the 1996 ACM SIGCOMM
Conference, Stanford, September 1996," and "RPC in the
10 x-Kernel: Evaluating New Design Techniques; In *Proc. of*
the Fourteenth ACM SYMP. on Operating Systems
Principles, pages 91-101, Asheville, NC, December 1993..
ILP encompasses optimizations on multiple protocol
layers. Much of the ILP tends to focus on integrating
15 data manipulations across protocol layers, but not on
optimizing control operations and message header
compression. On the other hand, ILP advantageously
compiles iteration in checksums, presentation
formatting, and encryption from multiple protocol layers
20 into a single loop to minimize memory references.
Currently, none of the Ensemble protocols touch the
application portion of messages. It would be desirable
to provide improved optimization techniques
incorporating the advantages of already developed
25 optimizations and focusing on such aspects of protocol
execution that are compatible with and orthogonal to the
existing optimization methods.

The above-described disadvantages of the previously developed optimization methods make it desirable to develop compilation techniques which make layered protocols execute as fast as non-layered protocols without giving up the advantages of using modular, layered protocol suites.

It is therefore an object of the present invention to provide a system and method which decreases actual computation and layering overhead in addition to latency and to provide optimization techniques applicable to a larger class of protocols.

It is yet another object of the present invention to provide optimized protocols of high performance which are easy to use. Normally, the protocol optimizations are made after-the-fact to already working protocols. This means that protocols are designed largely without optimization issues in mind. In the present invention optimizations require almost no additional programming, only a minimal amount of annotation of the protocol

5

10

15

20

25

30

30

Figure 6 is an illustration of a round-trip latency time line between two processes.

1. Layering Model

10

15

20

25

30

4. A Protocol stack 26 comprises protocol layers which are composed to create protocol stacks. A protocol stack is typically visualized as linear vertical stacks of protocols. Adjacent protocol layers communicate through two event queues, one for passing events from the upper

5. An Application 28 and a Network 30. The

15 6. A Scheduler 32 determines the order of execution of
 events in a protocol stack. The scheduler must ensure
 that events are passed between adjacent layers in the
 first-in-first-out order and that any particular
20 protocol layer is executing at most one event at a time.
 Also, all events must eventually be scheduled.

7. An Event trace 34 is a sequence of operations in a protocols stack. In particular, the term "event trace" is used to refer to the traces that arise in the normal case. Event trace 34 begins with the introduction of single event into protocol stack 26. The trace continues through the protocol layers, where other events may be spawned either up or down. In many cases even trace 34 may be scheduled in various ways. It is assumed that a particular schedule is chosen for a particular trace.

5

10

15

30

the bottom of the protocol stack. When the event emerges from the stack, network 30 transmits the message. The destination host inserts a *receive* event into the bottom of the protocol stack. Again, in a
5 simple scenario the event is repeatedly passed up to the top of the protocol stack and is handed to the application. In more complex situations, a layer can generate multiple events when it processes an event. For instance, a reliable communication layer may both
10 pass a *receive* event it receives to the layer above it, and pass an *acknowledgment* event to the layer below.

This model is flexible in that scheduler 32 has few restrictions on the scheduling. For example, the model admits a concurrent scheduler where individual layers
15 execute events in parallel.

The optimizations of the present invention were implemented as a part of the Ensemble communication system, which is described below. For an application builder, Ensemble provides a library of protocols that
20 can be used for quickly building complex distributed applications. An application registers 10 or so event handlers with Ensemble, and then the Ensemble protocols handle the details of reliably sending and receiving messages, transferring state, detecting failures, and
25 managing reconfigurations in the system. For a distributed systems user, Ensemble is a highly modular and reconfigurable toolkit. The high-level protocols provided to applications comprise stacks of small protocol layers. Each of these protocol layers
30 implements several simple properties: providing sets of high-level properties such as, for example, total ordering, security and virtual synchrony. Individual protocol layers can be modified or rebuilt to test with new properties or change the performance characteristics

00687439-101300

As an example, a type of event trace that occurs in many protocol stacks is considered. When there are no abnormalities in the system, sending a message through a protocol stack often involves passing a *send* event directly through the protocol stack from one layer to the next. If messages are delivered reliably and in correct order by the underlying transport, then the actions at the receiving side involve a *receive* event filtering directly up from the network, through the layers, to the application. Such an event trace is depicted in Fig. 3 at 34. Both the *send* and receive event traces are called linear traces because (1) they involve only single events, and (2) they move in a single direction either from network 30 to application 28 or vice versa through the protocol stacks.

For example, a *hierarchical routing protocol* is a protocol in which a broadcast to many destinations is implemented through a spanning tree of the destinations. As illustrated in Fig. 4,

a message is received from the network and passed to the routing layer. The routing layer forwards a copy down to the next destination and passes a copy to the network. The initiator sends the message to its neighbors in the tree, who then forward it to their children, and so on until it gets to the leaves of the tree which do not forward the message. Some of the traces in a hierarchical routing protocol would include the following steps, the first two of which are linear and the last step is non-linear:

1. Sending a message is a linear trace down through the protocol stack.

2. If a receiver is a leaf of the routing tree, then the receipt is a linear trace up through the stack.

5 3. If a receiver is not a leaf of the tree, the
 receipt will be a trace where: (1) the *receive*
 event is passed up to the routing protocol,
 (2) the *receive* event continues up to the
 application, and (3) another *send* event is
10 passed down from the routing protocol to pass
 the message onto the children at the next
 level of the tree, as shown in Fig. 4.

Determining and composing event traces is a procedure well suited for optimization. Determining event traces requires some annotation by protocol designers. They must identify the normal cases in the protocol layers, mark the conditions that must hold, and the protocol operations that are executed. Then the traces can be generated by composing the common cases across multiple layers. Note that entire layers are not being annotated and no additional code is being written: the annotation is done only for the common cases, which are usually a small portion of a protocol layer.

25 Intercepting event traces is an optimization
technique which is used after the event traces of a
protocol stack have been ascertained. After such time it
becomes possible to build alternative versions of the
code executed during those traces and modify the system
30 so that before an event is introduced into a protocol
stack, the system checks whether one of the event
conditions is enabled. If the event condition is not
enabled, then the event is executed in the protocol
stack in the normal fashion, and checking the conditions

[illegible]

has slowed the protocol down a little. If a trace condition holds, then the normal event execution is intercepted and instead the trace handler is executed. The performance improvement then depends on the percentage of events for which the trace condition is enabled, the overhead of checking the conditions, and how much faster the trace handler is.

The use of a trace handler assumes that there are no events pending in any of the intervening event queues. If there were a pending event, the trace handler would violate the model because the events in the trace would be executed out of order with regard to the previously queued event. The solution to this problem relies on the flexibility of the layering model, and works by using a special event scheduler that executes all pending events to completion before attempting to bypass a protocol stack, ensuring that there are no intervening events.

The transformation of the protocol stack maintains correctness of the protocols because trace handlers execute exactly the same operations as could occur in the normal operation of the protocol layers, ensuring, therefore, the soundness of the transformation. If the original protocols are correct, then the trace protocols are correct as well.

I2. Optimizing Event Traces

After event traces are determined and common paths of execution based on the event traces are identified the event traces are then optimized. The optimization

5

a.

10

20

25

30

5

20

The fourth step is to apply traditional optimizations to the trace handlers. This operation proves to be very effective, because the previous passes

5 marks an event record's field with some flag to cause an operation to happen at another layer, the flap can be propagated through the trace handler so that the flap is never set at the first layer or checked at the second layer.

10

15

25

30

These headers are compressed by our approach when they appear in the common path.

3. Non-constant headers include any other headers,
5 such as sequence numbers or headers used in negotiating reconfigurations. The non-constant headers are not compressed.

10 The above-described header compression optimizations are based on the use of connection identifiers, such as the ones described in U.S. patent application Serial No. 09/094,204, which is incorporated herein by reference. Connection identifiers are tuples containing addressing headers which do not change very
15 often. All the information in these tuples are hashed into 32-bit values which are then used along with hash tables to route messages to the protocol stacks. MD5 (a cryptographic one way hash function) is used to make hashing collisions very unlikely and other well-known
20 techniques can be used to protect against collisions when they occur. The use of connection identifiers compresses many addressing headers into a single small value. As a result, all subsequent messages benefit from such compression. Although the main goal of header
25 compression is to improve bandwidth efficiency, small headers also contribute to improved performance in transmitting the messages on the underlying network and in the protocols themselves because less data is being moved around.

30 In the present invention the concept of connection identifiers is extended to contain an additional field called the "multiplexing index." This field is used to multiplex several virtual channels over a single channel. Such use of connection identifiers allows

09637439.101300

The header compression optimization significantly reduces the header overhead of the protocol layers. Even though each of the constant headers is quite small, the costs involved in pushing and popping them becomes significant in large protocol stacks. In addition, by encoding these constant values in the trace code, standard compiler optimizations, such as constant folding and dead code elimination, are possible. For example, protocols in Ensemble have been successfully optimized using header compression. In many protocol stacks (including the ones with more than 10 protocol layers), traces often contain only one field. Without trace optimizations the headers with only one variable field add up to 50 bytes. With compression the total header size decreases to 8 bytes. 4 bytes of these 8 comprise a connection identifier. The other 4 bytes is a sequence number. Evidently, the compressed 8 byte header creates much less overhead in comparison with the headers in similar communication protocols, such as TCP (40 bytes or 20 bytes for TCP with header compression) Isis (over 80 bytes), and Horus (over 50 bytes).

Managing multiple formats is another task that can
30 be optimized. Two related problems arise when
additional header formats are introduced to protocol
stacks which expect only a single format. The first
problem occurs when a trace condition is not enabled for
a message received with compressed headers (for example,

00E10T" 6E128960

The third class of optimizations serves to improve latency of the trace handlers without decreasing the amount of computation. When a message is sent, there are certain operations (such as determining a message's sequence number) which must be executed before the message is transmitted, whereas some operations may be delayed until after the transmission (such as buffering the message). The effect of reordering operations is to decrease the communication latency. Similarly, some operations executed at the receiver are delayed until after the message is delivered.

Protocols are annotated to specify which operations can or cannot be delayed.

IV. Use of the ML Programming Language

The above-described optimization techniques were tested on the Ensemble system which is implemented entirely in the ML programming language. Ensemble is derived from a previous system written in C Horus, embodying numerous Horus features. The use of ML in Ensemble allowed to make all the structural changes that have improved performance. The optimizations provided by the present invention, Ensemble is much faster than Horus, even though C programs generally execute faster than ML programs. Ensemble benefits from a design which has tremendously improved performance, and the use of ML has been essential in being able to rapidly experiment and refine Ensemble's architecture in order to make these optimizations.

V. Implementation of the Optimizations

5

30

5
10

15

20

25

30